Lessons Learned from Adapting Aerospace Engineering Tools to the Parallel and Grid Computing Environment

Seungwon Lee,[1] Hook Hua,[1] Robert Carnright,[1] John Coggi[2] and David Stodden[2]
[1]Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91202
[2]Aerospace Corporation, 2350 E. El Segundo Blvd. El Segundo, CA 90245
Seungwon.Lee@jpl.nasa.gov
818-393-7720

Abstract— Many widely-used aerospace data visualization tools are interactive in nature and are programmed to run on a single processor. While such tools support real-time manipulation of simulation environments, the computations that generate the data are often batch oriented and computation intensive. In many cases, the data generation software is too tuned to a single-processor infrastructure to be readily adapted for emerging parallel and grid computing environments. This paper presents several lessons learned from adapting an aerospace engineering tool to the parallel and grid computing architecture. The architecture provides the ability to perform high-power computing by distributing process execution across many computers connected by a dedicated network or the internet. Some of the challenging tasks involved in the adaptation are (1) to decouple the user interface and display functions from the computational functions, since interaction and graphics are usually unnecessary expenses in parallel and grid computing, (2) to identify and parallelize computationally expensive functions without the drastic modification of the code and data structures, (3) to find a lightweight, yet versatile software solution for a client-server machine interface for remote job execution. The solutions we found for these elaborate tasks are presented and their pros and cons are discussed.[1] [2]

TABLE OF CONTENTS

1. INTRODUCTION 1
2. PARALLEL COMPUTING 1
3. GRID COMPUTING 3
4. CONCLUSIONS 4
REFERENCES 4
BIOGRAPHY 4

1. INTRODUCTION

As the computational resources and processing power become readily available, science and engineering tools are being adapted to take advantage of parallel and grid computing. These computing methods provide improved performance by distributing process execution across many computers connected by a dedicated network or the internet [1,2]. This paper presents lessons learned from adapting an aerospace engineering tool called Satellite Orbit Analysis Program (SOAP) to the parallel and grid computing environment.

SOAP is a widely used tool within the aerospace community for the visualization and analysis of space missions. Initially, all of the software ran in single processor mode. However, mission trade studies involve multiple independent SOAP runs, and the total CPU time required for lengthy missions can reach over one thousand CPU hours. The high computational requirement prohibited Team X at JPL and the Concept Design Center (CDC) at The Aerospace Corporation from performing trades and coverage analyses in a concurrent engineering environment. We have developed parallel computing and remote execution capabilities in SOAP in order to reduce the per-study duration and cost of mission trade studies. This is the topic covered by the rest of this paper.

2. PARALLEL COMPUTING

As a prerequisite to the parallel computing development, we developed a version of SOAP that can be configured to execute in either parallel or interactive mode. In the parallel mode, the user interface and the display are suppressed, and only the computational components are executed as a batch job on the server. The results are then conveyed back to the client machine, where they may be interactively viewed by a client machine version, which employs the user interface and visualization components. The decoupling of graphics and visualization eliminates their unnecessary expense of computational power in the parallel and grid computing environment. In the graphics-free batch-mode executable on the server, the graphical and user-interface functions have been mapped to empty stub functions. This allows the domain-specific application code to appear the same while it simply linked to null library functions having the same Application Programmer's Interface (API).

The parallel computing development starts with identifying the computational components in the code that can benefit the most from parallelization. The ideal candidates are those that are easy to partition into quasi-independent executions. For example, a loop that calculates separate function evaluations on independent variables is ideal. The development of parallelization of the loop involves a

1

[1] 1-4244-0525-4/07/$20.00 ©2007 IEEE.
[2] IEEEAC paper #1067, Version 1, Updated October 27, 2006

partitioning of the loop into independent jobs and the collection of the job results at the end. If the partitioned jobs are coupled, the result of a partition needs to be sent to other partitions that have the coupling with the sender. The message passing between the processors due to the job coupling and synchronization unavoidably reduces the parallel speed-up. Therefore, the less coupled the partitioned job becomes, the more the overall job performance improves.

Another important feature to look for in the identification of the parallel computation component is computational intensity. It is ideal to parallelize the most computationally expensive parts, because these dominate the execution time. For example, if one part takes *W* fraction of the total computation time and is parallelized up to the ideal limit, the speed-up of the total computation time with *n* number of processors will have the upper limit given by

$$\text{speedup} = \frac{1}{(1-W) + W/n} \qquad (1)$$

The equation shows that as *W* becomes close to 1 the speed-up approaches n. Conversely, as *W* becomes close to 0 the speed-up approaches 1. Therefore, the parallelization speed-up is greatly determined by *W* the computation time of the parallelized part in terms of the fraction of the total computation time.

Guided by these heuristics, we have selected two routines in SOAP as the first targets for parallelization. They are "Parametric Study" and "Contour Calculation" modules. The associated routines compute one or more analytical functions in a loop and become quickly computationally intensive as report duration, time resolution, and variable complexity increase. The parametric study is a loop in a variable domain while the contour calculation is a loop in a spatial domain. The result of each variable and spatial coupling is completely independent, so that message passing is not needed in the parallel execution of the loop.

The variable and spatial domains in the two functions are sliced and distributed to processors proportional to their processing powers. Faster computers are assigned more jobs than slower computers. This job distribution scheme is essential when a cluster computer is composed of heterogeneous computers with different processing powers. The result of the partitioned computation is collected at the end of the computation. A single processor handles the task of writing the numeric results to a file. Minimal message passing ensures the faster execution of the parallel implementation.

The message passing between processors and the synchronization of the parallel execution are handled by Message Passing Interface (MPI) library, the most commonly used library for parallel computing in a distributed memory architecture [3]. Only a few key functions in MPI are used for the present work. The MPI_Bcast function is used for the result collection after the parallel execution of the loop is complete. The MPI_Barrier function is used between the end of the parallel execution and the beginning of the local result collection so that message passing is not timed out by a processor which is delayed in finishing the parallel job. The MPI_Send and MPI_Recv functions are used in place of the MPI_Bcast when only one processor is required to have the complete array of results and write out the results.

The two parallel routines are tested on JPL's Dell Cluster Computer to benchmark the performance. The cluster computer is composed of 1024 Intel Pentium 4 Xeon processors with 3.2 GHz clock speed, 1MB cache, 32-bit integer, and 2 GB memory per CPU. Each processor is connected by Myricom Myrinet-2000 fibre interconnect. It can provide 6.55 TFLOPS at peak.

The parallel parametric study routine is tested with the study in a 50x50x50 variable grid that determines the best combination of right-ascending node and solar panel orientation for a Mars orbiter that would maximize the intensity of sunlight on the panels for a given 3-day period. Figure 1 shows the performance of the study with respect to number of processors used. The speed-up is given by the ratio of the computation time on single processor to the time on multiple processors. The speed-up is almost linear to the number of processors up to 32, and the speed-up of 234.3 is obtained with 256 processors. The robust speed-up is due to the fact that most of the computation time is spent on the parallel computation portions of the code as opposed to those based on sequential execution.

The parallel contour calculation routine is also tested on JPL's Dell Cluster Computer. The test example is to calculate the fraction of time at least one of two satellites in elliptical orbits is visible from the Earth for 24 hours over a 120x120 planetary surface grid. Figure 2 shows the performance of the study with respect to the number of processors used. The improvement in performance is less ideal than the parametric study example. The weaker improvement is due to the short computation time in parallel. In this case, the amount of non-parallel calculations portions, such as initial setup and final result collection is approaches that of which can be executed in parallel. We expect that when the study involves longer periods of parallel computation, the speed-up will improve. This would happen if the time domain or sampling rate of the study was increased

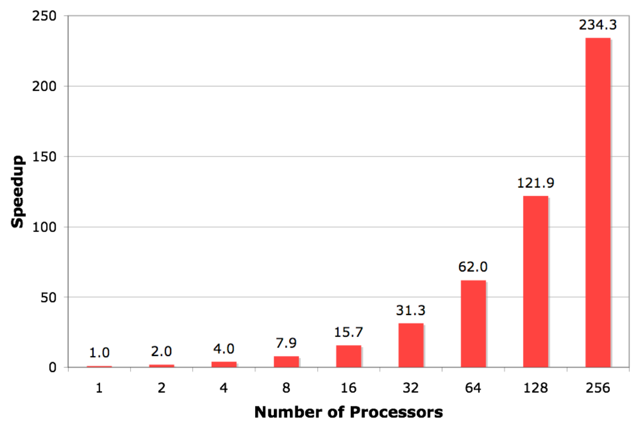


Figure 1. Speed-up of the parallel parametric study routine on JPL's Dell Cluster Computer.

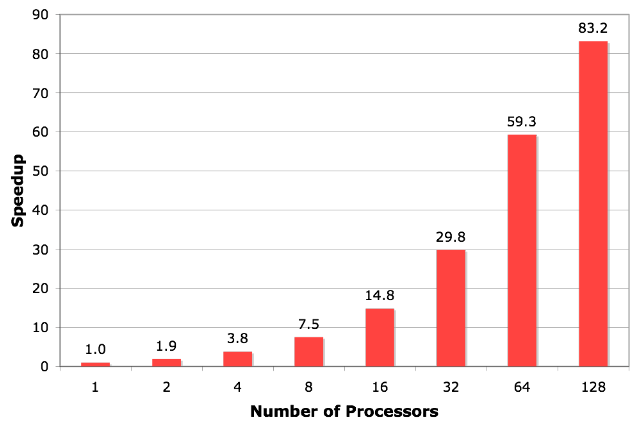
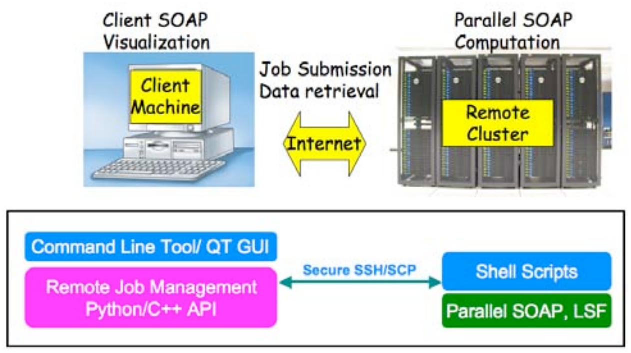


Figure 2. Speed-up of the parallel contour calculation routine on JPL's Dell Cluster Computer.

3. GRID COMPUTING

The notion of grid computing, in a broad sense, is to use the resources of many separate computers connected by a network. In the context of the present work, this involves the use the remote cluster resource from a user client machine through a lightweight tool. For the SOAP effort, the remote job manager had to satisfy the following requirements: (1) support secure password and data transport, (2) be compatible with SOAP Windows, Macintosh OS X, and Linux client platforms.

In compliance with these requirements, we have developed a remote job manager for the user interface between the parallel SOAP on a remote cluster computer and the serial SOAP on a user client machine. The manager handles a job submission, job query, job killing, job result retrieval, and job cleaning upon a user's demand. The manger is written in Python and uses OpenSSH for the secure connection between two machines. Both Python and OpenSSH are available for all four major client platforms and are open-source products. Furthermore, both programs are well maintained by a large community, providing a reliable software platform for future development.



Client SOAP Visualization
Client Machine
Job Submission
Data retrieval
Internet
Parallel SOAP Computation
Remote Cluster
Command Line Tool/ QT GUI
Remote Job Management Python/C++ API
Secure SSH/SCP
Shell Scripts
Parallel SOAP, LSF

Figure 3. Overall architecture of the remote job manager for the execution of the parallel SOAP in the grid computing environment.

In the manager, process execution and error handling is controlled by Pexpect, which is a Pure Python module for spawning child applications such as ssh, ftp, passwd, telnet, controlling them, and responding to their output and errors [4]. One of the important tasks done via Pexpect is to securely send the password to ssh and scp for remote login and remote file transfer, respectively. For security reasons, OpenSSH's ssh and scp commands both rely on the POSIX TTY terminal to accept passwords rather than STDIN. Pexpect supports this security behavior. Additionally, because Pexpect is implemented as a Python package, the functionality of sending passwords is done entirely within the same process-space as the host process that embeds and/or imports the Pexect module. OpenSSH's reliance on the TTY terminal however becomes a drawback since Pexpect has to be run from within a POSIX compliant environment such as Linux and MAC OS X. For Windows, a Python interpreter alone is insufficient. Python must be run from within Cygwin which is the POSIX port for Windows. Cygwin provides a Linux-like environment and provides many Unix API functionalities [5].

The remote execution of SOAP is initiated by transferring an input file from the user machine to remote machine using the scp command and executing a remote shell script on the cluster computer using the ssh command. The remote shell script first processes the input argument from the user machine, edits the user-provided input file to make it compatible with the parallel (server) version of SOAP, and then initiates a job submission by invoking the command of the local scheduler (LSF on JPL's Dell Cluster computer). After the job submission, the user can query whether job is running properly or has been completed. When the job is completed, the results can be retrieved by using the same ssh and scp command mechanisms as for the job submission. All the commands are integrated into Python module so that the user does not have to directly use the ssh and scp commands. Figure 3 summarizes the architecture of the developed remote job manager.

The job manager is tested on Unix, MAC OS X, and Windows machines. In Unix and MAC OS X machines, OpenSSH and Python are often already installed. The installation of the remote job manager is simply to compile the python scripts we have written. For Windows machine, Cygwin environment is used to install OpenSSH commands and Python.

As alternatives, we have also considered other approaches such as (Simple Object Access Protocol (SOAP)/XML web services approach [6] and Globus Toolkit [7]. While these approaches are more versatile than OpenSSH and Python, the libraries and tools that support these approaches are not as standardized. They require the development of specific C/C++ APIs to couple with the web services and Globus libraries. Especially, the Globus libraries are huge (over 1 GB) even with the minimum set of files, making the porting of Globus-based applications to a client machine difficult. Some of Globus libraries such as GRAM are currently incompatible with Windows platform. The libraries for SOAP/XML web services and Globus are expected to mature and become more platform-independent in near future. Therefore, these alternative approaches provide a long-term solution for a versatile remote job and resource allocation manager.

Another reason we decided not to use Globus was a direct result from policy restrictions of the cluster we were using. For security reasons, the cluster does not permit daemons to be installed and run directly from the front-end node of the cluster. The Globus approach would have required setting up some http daemon such as Apache Tomcat for Web Services, GridFTP for file transfer, etc. on the cluster. Additionally, the client-side file size and dependency requirements of Globus 4 were not favored by the customer requirements. The server-side installation requirements with the Globus approach would have violated policy restrictions of the cluster deployment.

By developing a remote job management API that leverages OpenSSH, we were able to satisfy the light-weight requirements on the client-side, as well as the server-side restrictions of no additional services or daemons installed. The functionality of the remote job management API requires only ssh and scp interaction with the cluster—which is a status quo standard.

This approach also permits future enabling of Globus or other simpler Web Services to the restricted cluster by exposing the remote job management API as a service on another machine. This allows the cluster to remain secure, while another machine that uses the remote job management API can act as a Web Services provider. All requests to this machine will be delegated to the cluster via the job management API.

4. CONCLUSIONS

We have adapted the SOAP aerospace engineering application to the parallel and grid computing environment. Two computationally intensive capabilities are parallelized using the MPI library. In the spirit of grid computing, a lightweight tool for remote job management has been developed. The parallelized SOAP demonstrates the speed-up that is almost ideal up to 64 processors. The remote job manager allows users to execute a job using the parallel version of SOAP on a remote cluster computer, to monitor job status and to retrieve job results.

REFERENCES

[1] "Introduction to Parallel Computing," Blaise Barney, Online Tutorial, http://www.linl.gov/computing/tutorials/parallel_comp/

[2] http://en.wikipedia.org/wiki/Grid_computing/

[3] http://www-unix.mcs.anl.gov/mpi/

[4] http://pexpect.sourceforge.net/

[5] http://www.cygwin.com/

[6] http://en.wikipedia.org/wiki/SOAP/

[7] http://www.globus.org/tookkit/

BIOGRAPHY

Seungwon Lee is a senior staff member of the High Capability Computing and Modeling Group at the Jet Propulsion Laboratory. Her recent work includes evolutionary optimization application, high performance computing, materials modeling, and nonlinear dynamics system. Her work is documented in numerous journals and conference proceedings. She has a B.S. and M.S. in Physics from Seoul National University in Korea, and has a Ph.D in Physics from Ohio State University.

Hook Hua is a staff member of the High Capability Computing and Modeling Group at the Jet Propulsion Laboratory. His recent work includes constrained optimization and analysis with binary integer programming and genetic algorithms, knowledge base expert systems with natural language processing, and image processing models.

He is also experienced with distributed architectures and middle-ware frameworks. He has a B.S. in Computer Science and a B.S. in Applied Mathematics both from University of California, Los Angeles.

John Coggi is a senior engineering specialist in the Computer Systems Division of The Aerospace Corporation. He is one of the primary developers of the Satellite Orbit Analysis Program (SOAP) and uses the tool to perform advanced orbital analysis for various customers. He is experienced in Macintosh software development, MPI programming, and web development. He has a B.S. in Chemistry from the University of California, Irvine and an M.S. in Computer Science from California State University, Long Beach.

David Stodden is a senior project leader in the Computer and Software Division of The Aerospace Corporation. He is the project lead for the Satellite Orbit Analysis Program (SOAP) software and is an AIAA Associate Fellow. He specializes in orbit analysis, 3D visualization, and MS-Windows based software development. He holds a M.S. in Computer Engineering and a B.S. in Computer Science from California State University, Long Beach.